There are six problems labeled A-F. The first three problems are about the quotient operation on languages. The last three problems are about the relationship between DFAs and NFAs. Depending on your background, you might find one block of problems easier than the other.

See Problem 1.45 in our textbook for the definition of the quotient A/B. This operation is quite confusing. It recurs a few times during the course. Problem A deals with a special case. Problem B gives you an idea of why it is a "quotient". Problem C is more difficult.

A. If $A = \Sigma^*$, then what is A/B? [Hint: There are two possibilities, depending on B.]

B. Find an example of two infinite languages A and B such that $(A/B)B = A$. Also find an example of two languages A and B such that $(A/B)B \neq A$.

C. Do problem 1.45. Your solution will probably involve some kind of description of a DFA or NFA for A/B. Your solution should explain why the automaton accepts all strings in A/B and why it rejects all strings that are not in A/B.

This first DFA-NFA problem is mechanical.

D. Do problem 1.16b in our textbook.

The remaining problems deal with Python implementation of NFAs without ϵ-transitions. We need a way to represent them in code. Let's agree that they're just like DFAs, except in the final element δ of the tuple. Just as in DFAs, δ takes as input a state and an alphabet symbol. What's different is that, as output, δ returns a set (tuple) of states rather than a single state.

E. Write a Python function `nfa`. It is the sibling to your function `dfa` from the previous assignment. It takes two inputs — an NFA without ϵ-transitions and an input string — and returns whether the NFA accepts the input string. (My solution is 12 lines of code including comments.) Include demo code and show the results of the demo code, so that a grader can assess whether your code works. Print everything on paper.

F. Write a Python function `dfaFromNFA`. This function takes as input an NFA without ϵ-transitions. It returns an equivalent DFA built using the power set construction. (My solution is 14 lines of code including comments. It uses the following helper functions.) Include demo code and its results, on paper.

```
# X and Y are sets. Returns the set X intersect Y. Assumes that the elements
# are 'simple enough to be compared naively'. The output is ordered to match
# the order in X.
def setIntersection(x, y):
    result = []
    for z in x:
        if z in y:
            result.append(z)
    return tuple(result)

# X is a set. Returns the power set P(X). The output set itself is not sorted
# in any particular way, but each element of the output set is sorted to match
# the order in X.
def setPower(x):
    if len(x) == 0:
        return ((),)
    else:
        withoutFirst = setPower(x[1:])
        withFirst = [(x[0],) + y for y in withoutFirst]
        return tuple(withFirst) + withoutFirst
```

Let me tell you about a pitfall, that you might not see immediately. Your DFA states are sets of NFA states. These sets are represented as tuples. Tuples are ordered, whereas sets are not. For example, the tuples `(1, 3, 8)` and `(8, 1, 3)` represent the same set, but Python does not regard them as equal. So you need to pay attention to the order of elements within certain sets. That's why the documentation for the helper functions above talks so much about order.